1. Round Robin Scheduling

#include <stdio.h>

void roundRobin(int processes[], int n, int burst\_time[], int quantum) {

int remaining\_time[n], waiting\_time[n], turnaround\_time[n];

int total\_wt = 0, total\_tat = 0;

int t = 0; // Current time

for (int i = 0; i < n; i++)

remaining\_time[i] = burst\_time[i];

while (1) {

int done = 1;

for (int i = 0; i < n; i++) {

if (remaining\_time[i] > 0) {

done = 0; // There is still a pending process

if (remaining\_time[i] > quantum) {

t += quantum;

remaining\_time[i] -= quantum;

} else {

t += remaining\_time[i];

waiting\_time[i] = t - burst\_time[i];

remaining\_time[i] = 0;

}

}

}

if (done)

break;

}

for (int i = 0; i < n; i++)

turnaround\_time[i] = burst\_time[i] + waiting\_time[i];

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

for (int i = 0; i < n; i++) {

total\_wt += waiting\_time[i];

total\_tat += turnaround\_time[i];

printf("P%d\t%d\t\t%d\t\t%d\n", processes[i], burst\_time[i], waiting\_time[i], turnaround\_time[i]);

}

printf("\nAverage Waiting Time: %.2f", (float)total\_wt / n);

printf("\nAverage Turnaround Time: %.2f\n", (float)total\_tat / n);

}

int main() {

int n, quantum;

printf("Enter the number of processes: ");

scanf("%d", &n);

int processes[n], burst\_time[n];

printf("Enter the burst times of the processes:\n");

for (int i = 0; i < n; i++) {

processes[i] = i + 1;

printf("P%d: ", i + 1);

scanf("%d", &burst\_time[i]);

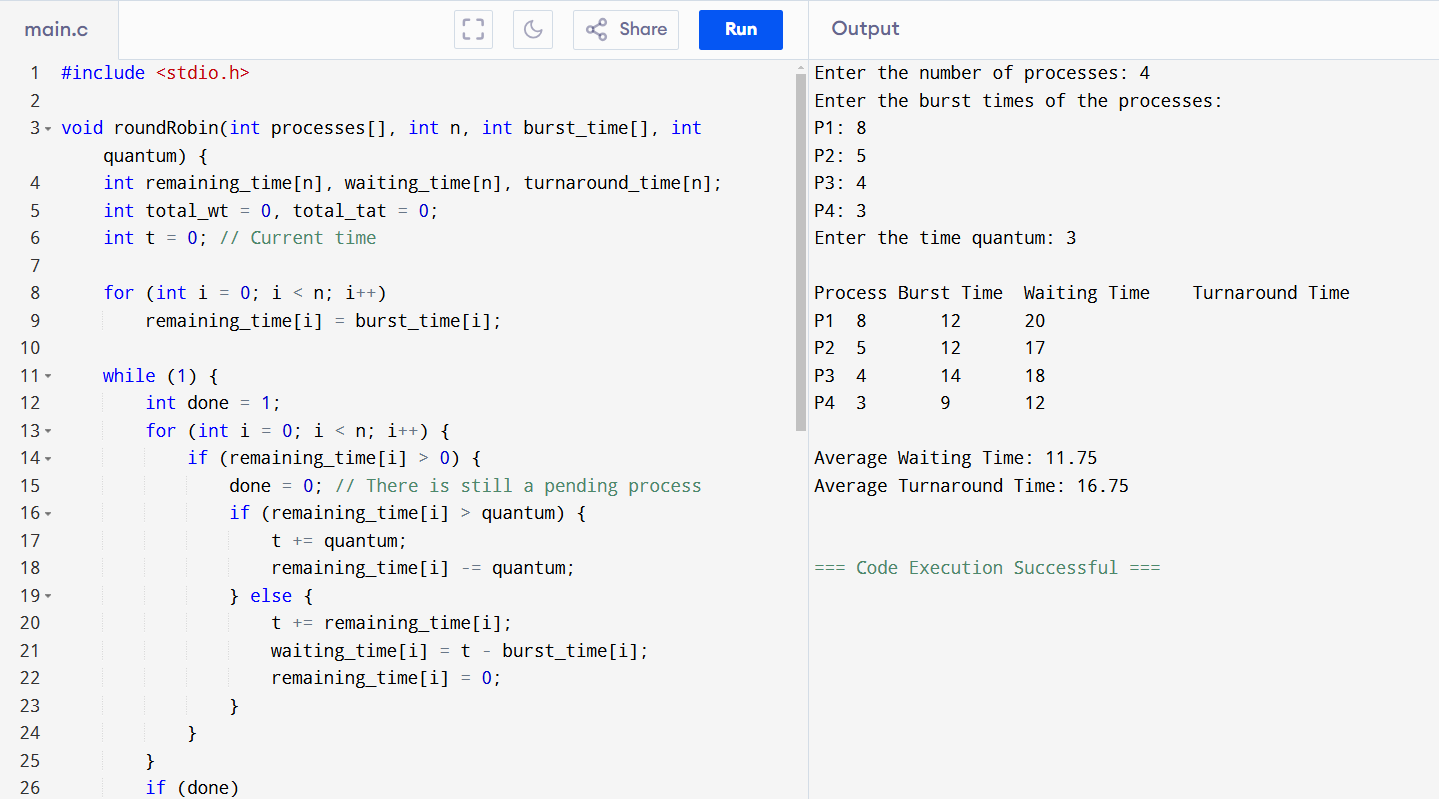
}

printf("Enter the time quantum: ");

scanf("%d", &quantum);

roundRobin(processes, n, burst\_time, quantum);

return 0;

}

1. Inter Process Communication

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

struct msg\_buffer {

long msg\_type;

char msg\_text[100];

};

int main() {

key\_t key;

int msgid;

struct msg\_buffer message;

key = ftok("msg\_queue", 65);

msgid = msgget(key, 0666 | IPC\_CREAT);

message.msg\_type = 1;

strcpy(message.msg\_text, "Hello from IPC message queue!");

msgsnd(msgid, &message, sizeof(message), 0);

printf("Message sent: %s\n", message.msg\_text);

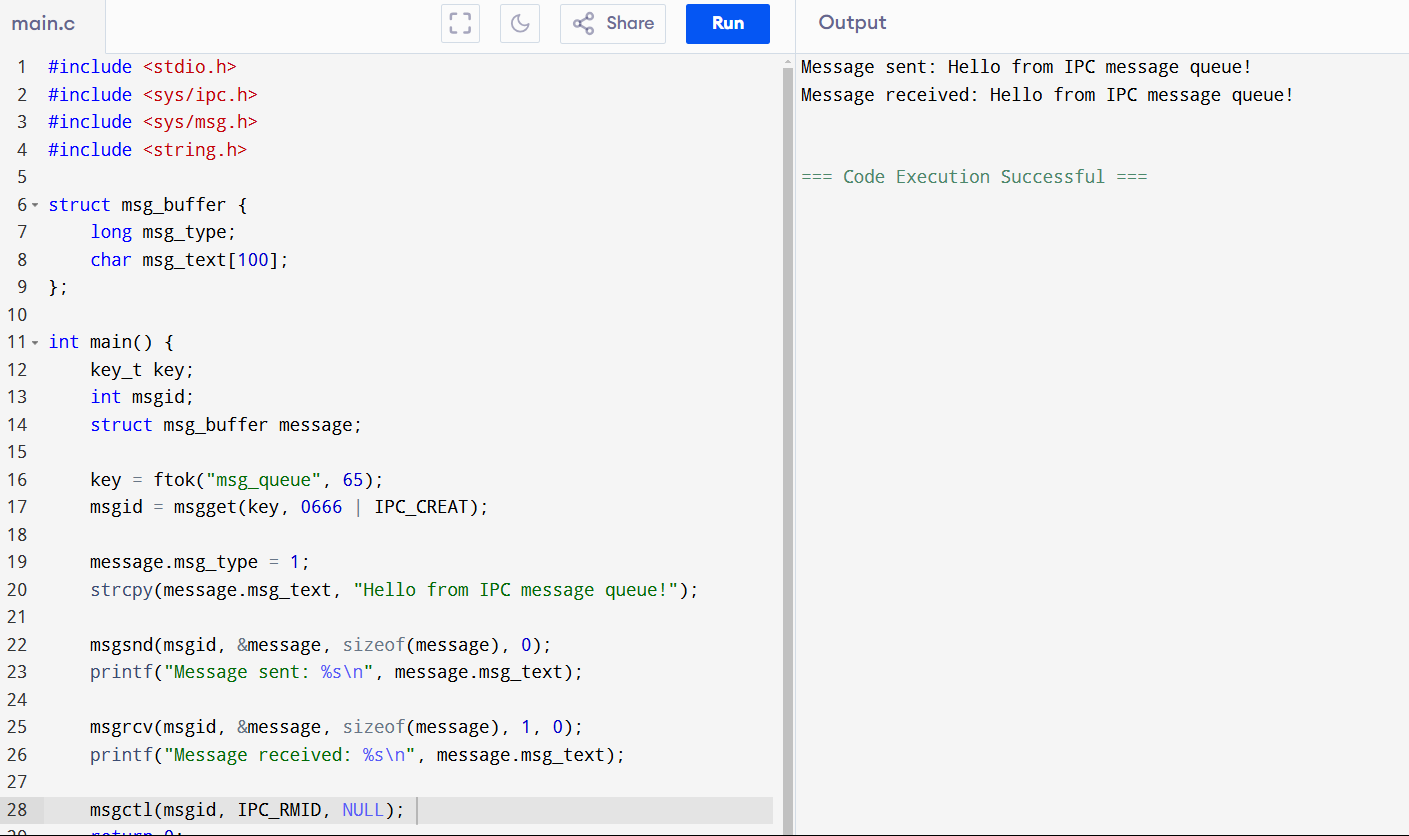
msgrcv(msgid, &message, sizeof(message), 1, 0);

printf("Message received: %s\n", message.msg\_text);

msgctl(msgid, IPC\_RMID, NULL);

return 0;

}



1. Dining-Philosophers problem

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define N 5

sem\_t forks[N];

pthread\_mutex\_t mutex;

void think(int philosopher) {

printf("Philosopher %d is thinking...\n", philosopher);

sleep(1);

}

void eat(int philosopher) {

printf("Philosopher %d is eating...\n", philosopher);

sleep(2);

}

void \*philosopher(void \*arg) {

int id = \*(int \*)arg;

while (1) {

think(id);

sem\_wait(&forks[id]);

sem\_wait(&forks[(id + 1) % N]);

pthread\_mutex\_lock(&mutex);

eat(id);

pthread\_mutex\_unlock(&mutex);

sem\_post(&forks[id]);

sem\_post(&forks[(id + 1) % N]);

}

}

int main() {

pthread\_t threads[N];

int philosophers[N];

pthread\_mutex\_init(&mutex, NULL);

for (int i = 0; i < N; i++)

sem\_init(&forks[i], 0, 1);

for (int i = 0; i < N; i++) {

philosophers[i] = i;

pthread\_create(&threads[i], NULL, philosopher, &philosophers[i]);

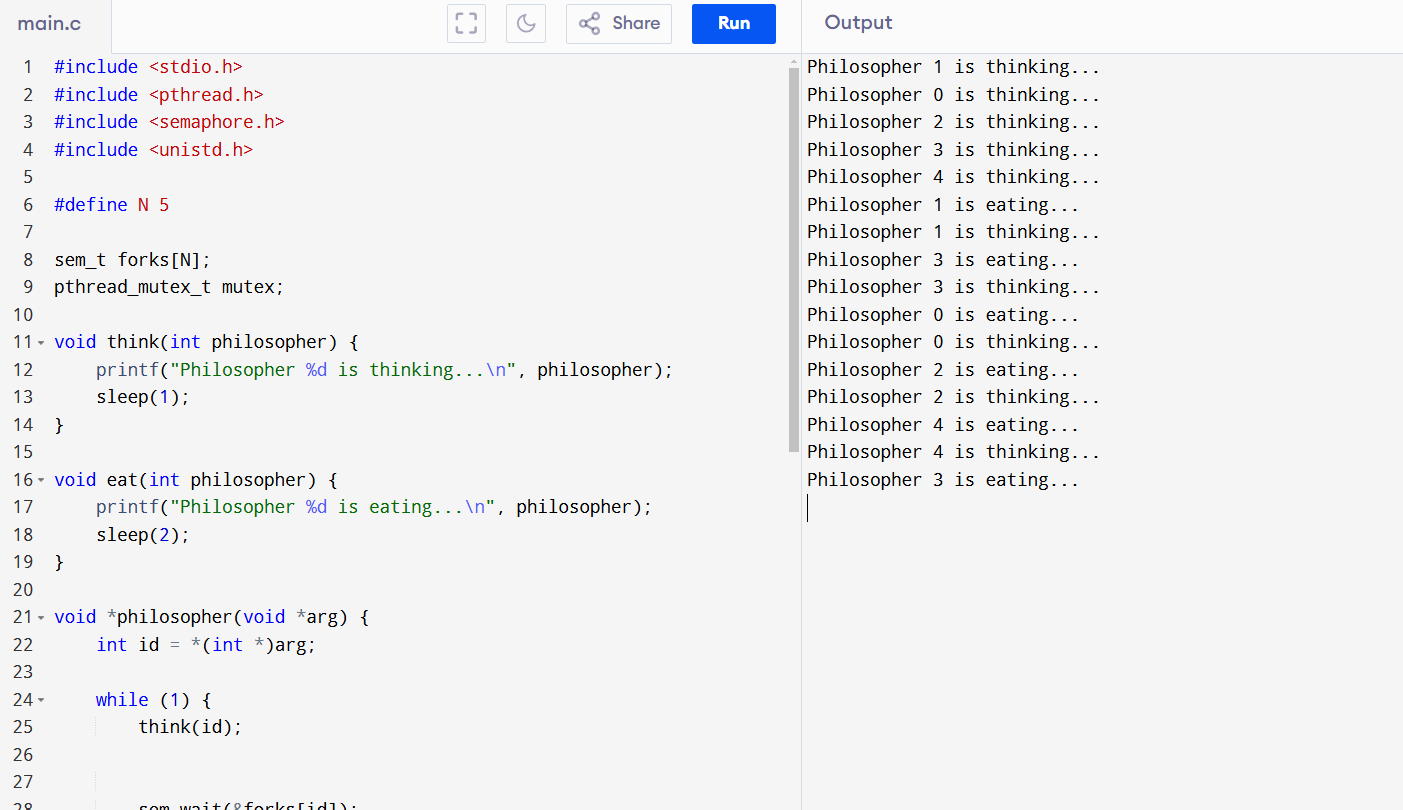
}

for (int i = 0; i < N; i++)

pthread\_join(threads[i], NULL);

return 0;

}



1. Banker’s algorithm

#include <stdio.h>

#define P 5

#define R 3

void calculateNeed(int need[P][R], int max[P][R], int allocation[P][R]) {

for (int i = 0; i < P; i++)

for (int j = 0; j < R; j++)

need[i][j] = max[i][j] - allocation[i][j];

}

int isSafe(int processes[], int available[], int max[P][R], int allocation[P][R]) {

int need[P][R], work[R], finish[P] = {0}, safeSeq[P], index = 0;

calculateNeed(need, max, allocation);

for (int i = 0; i < R; i++)

work[i] = available[i];

for (int count = 0; count < P; count++) {

int found = 0;

for (int i = 0; i < P; i++) {

if (finish[i] == 0) {

int flag = 1;

for (int j = 0; j < R; j++)

if (need[i][j] > work[j]) {

flag = 0;

break;

}

if (flag) {

for (int k = 0; k < R; k++)

work[k] += allocation[i][k];

safeSeq[index++] = i;

finish[i] = 1;

found = 1;

}

}

}

if (!found) {

printf("System is in an UNSAFE state! Deadlock may occur.\n");

return 0;

}

}

printf("System is in a SAFE state.\nSafe Sequence: ");

for (int i = 0; i < P; i++)

printf("P%d ", safeSeq[i]);

printf("\n");

return 1;

}

int main() {

int processes[P] = {0, 1, 2, 3, 4};

int available[R] = {3, 3, 2};

int max[P][R] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

int allocation[P][R] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

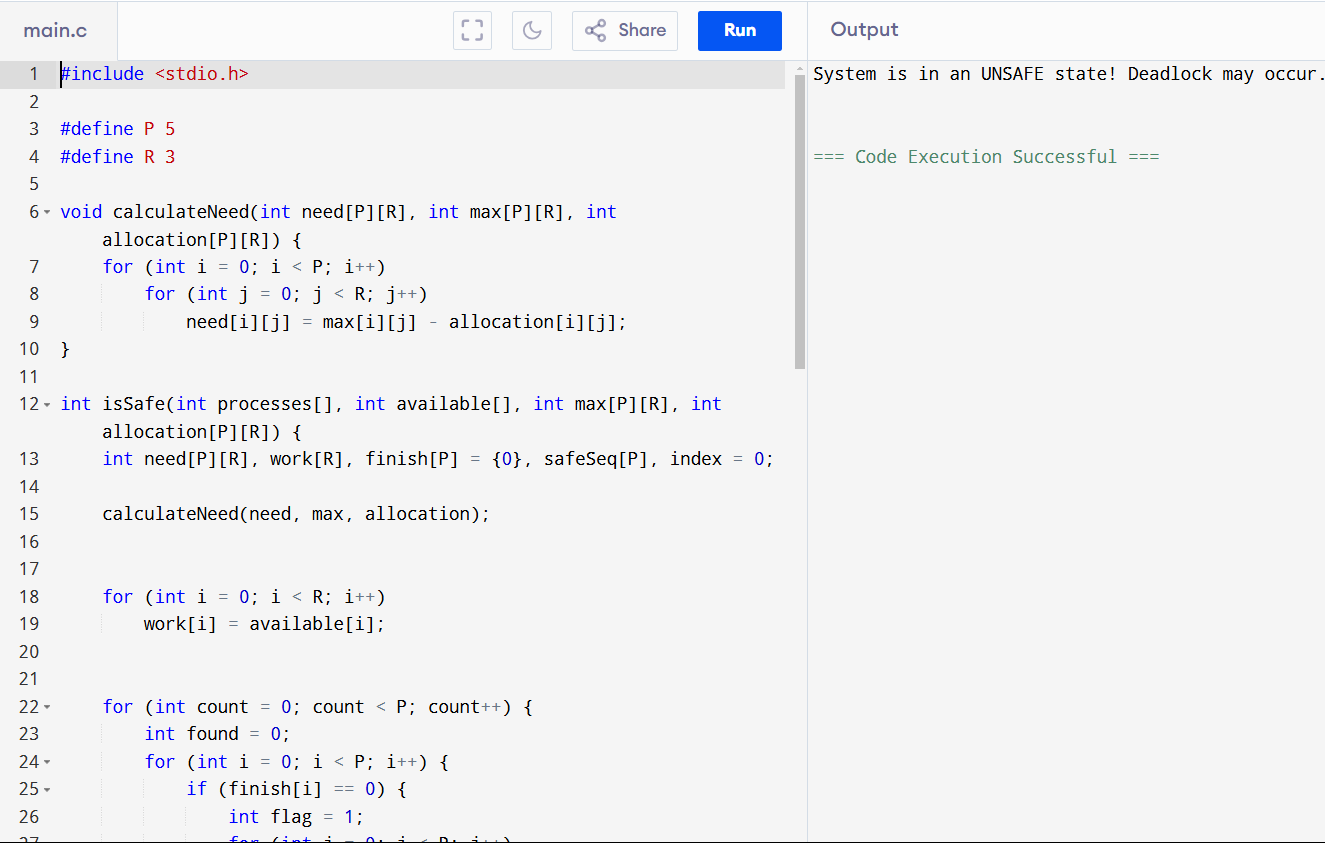
{0, 0, 2}

};

isSafe(processes, available, max, allocation);

return 0;

}



1. Producer Consumer Problem

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty, full;

pthread\_mutex\_t mutex;

void \*producer(void \*arg) {

int item;

for (int i = 0; i < 10; i++) {

item = i + 1;

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[in] = item;

printf("Producer produced: %d\n", item);

in = (in + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

sleep(1);

}

}

void \*consumer(void \*arg) {

int item;

for (int i = 0; i < 10; i++) {

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item = buffer[out];

printf("Consumer consumed: %d\n", item);

out = (out + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

sleep(2);

}

}

int main() {

pthread\_t prod, cons;

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

pthread\_mutex\_init(&mutex, NULL);

pthread\_create(&prod, NULL, producer, NULL);

pthread\_create(&cons, NULL, consumer, NULL);

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

